Enterprise-Wide Logging Through Software Life-Cycle

Sharil Tumin[1] and Sylvia Encheva[2]

**[1] IT Department, University of Bergen,
Bergen, 5020, Norway**

**[2] Faculty of Technology, Business and Maritime Education, Stord/Haugesund University College
Haugesund, 5528, Norway**

Abstract

A software has distinctive phases; design, development, testing, deployment and retirement. Logging helps developers, testers, maintainers, and managers to effectively acting, reacting, and interacting with the planned and unplanned events during a program execution. A software gets input from and puts output to its dynamically changing operating environments. Logging helps to record historical events to be used for error-correction, capacity planning, and general value added amendments to its functions.

Keywords: *Software engineering, events logging, proactive modeling, reactive planning, distributive collaborative reporting.*

1. Introduction

Computer programs P are designed, developed, and maintained to provide solutions S for some very specific problems Q within their executing environments E, in another words, $S = P(Q \in E)$. More often than not, these environments are changing constantly. What is more difficult to developers and maintainers of computer programs is that these changing parameters are in fact constraints that define initial functions and operational boundaries of these programs.

Many large software projects never see the light of day. They are often abandoned before completion being unable to meet the demand of ever changing operational environments in which these softwares were meant to be deployed. The more complicated these projects are the more exposed they are to the risk of change. A number of methodologies are introduced to tackle the complexities of a software project: objects oriented programming, modulars base libraries, flexible multi-version supporting packages, agile development methods, piecewise deployment, and usage feedback iterative functional increments [1].

In this paper we are concern with logging planned and unplanned events from some computer programs under execution. There are other types of data-loggers for special purpose data gathering applications such as weather data of temperature, atmospheric pressure, humidity and other data pertinent to weather reports. Also, in computer systems loggers are used for 1) audit, 2) transaction, 3) intrusion, 4) connection, 5) access, 6) activity, and 7) alert [2].

Here, we are focused on the type of loggings closely related to the processes of a software life-cycle. While the main aims of data-loggers, i.e. weather data, is to provide better weather predictions, events loggings in computer programs intent to provide end-users with the best possible information services.

2. Software Logging and Life-Cycle

What is in common and useful in the phases of a software life-cycle of 1) Design, 2) Develop and 3) Deploy, is without doubt logging. A *log* is meant here as a unit of timestamped recorded message with relevant information concerning a specific event, whether planned or unplanned. Wedge between Design and Develop, and between Develop and Deploy is Test. Testing is a process by which we *measure* the difference between what actually is achieved and what were the desired goals, where here a carefully designed logging methodology can be an invariably useful tool. As such logging is not just storing and reporting events but more of an awareness tool through distribution and collaboration, see Fig. 1.

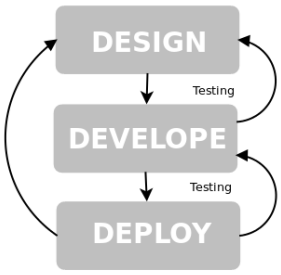

Fig. 1 Design, Develop, Deploy

A designer team, in collaboration with applications operational managers, can make use of applications logs about usage and errors of previous or similar programs to design a newer and better version of programs to meet new emerging requirements. A group of developers working with designers can use logs to check at the early stage the correctness of assumptions about constrains and protocols in the specifications. Operational managers and working with the developers now acting as maintainers use logs for bug fixes and capacity planning. As the operational environments are dynamically changing, logs can be used to record exceptions due to changes outside the control of application operational managers.

A software has several distinctive stages; 1) design, 2) development, 3) testing, 4) deployment, and 5) retirement. The degree of involvement of different stake-holders depends on stages on which the software is currently in. Ideally at stage 4), the software is at a steady state of $S = P(Q \in E)$. The solutions S provide balance nicely to the problems Q the program P was designed and developed for within its operational environment E . In reality E is not static but dynamically changing, reacting to outer and greater environments encompassing it.

Stages 1), 2) and 3) can have many iterations, driven by issues found in 4). In stage 4) provision of pertinence information needs to be provided for feedback to 2) for error-corrections and feedback to 1) for value added amendments to its functions. These feedbacks must be found in the loggs. With these views in mind, logging has to be proactive in its modeling and reactive in its operative planning.

Even if the solutions S are fixed in direct relation to the problems Q , the programs P must necessarily change with changing environments E . When we look at different minor versions of a software we see its time progression as

$$S = [P_{t1}(Q \in E_{t1}), P_{t2}(Q \in E_{t2})...P_{tn}(Q \in E_{tn})]$$

Major versions software releases R introduce new solutions S_{tm} to new problems Q_{tm} within the same environment or an extended operational environment E^{+}_{tm} ; $R = [S_{t1}, S_{t2}...S_{tn}]$. A software is in passive state when there are no new major releases, and not supported when there is no minor releases. After a certain period of inactivity the software will not be considered as useful and enter the retirement stage and eventually no longer be in deployment. A program is considered useful as long as it is in stage 4). There are many reasons why a program can not forever be in stage 4), one common reason is that the program had become to complicated to be maintained. The central tenet of usefulness is simplicity.

2.1 Data and Events-Loggers

The main difference between a special purpose data-loggers and computer application events-logger is that the former has definite units of data to collect with fixed format, while for the later no such presumptions can be arrogated. Computer programs or application events-loggers at best can be said as planned chaos, since there is no standard; 1) format, 2) schema, 3) taxonomy, 4) transport and 5) API (application programming interface). Developers are free to choose whatever works on the base that there is no; 1) standard logging guidance, and 2) common knowledge of what and how to log. Logging frameworks are here to mitigate the chaos or at least try to introduce some of the elements of best practice in logging.

Unix/Linux Syslog system, was developed in the 1980s by Eric Allman as part of the Sendmail project [3], providing us with some standard logging guidance on how to manage OS (Operating System) properly by leveraging logging of events. Units of log (or logging massages) refer to a facility; 1) *auth*, 2) *authpriv*, 3) *daemon*, 4) *cron*, 5) *ftp*, 6) *lpr*, 7) *kern*, 8) *mail*, 9) *news*, 10) *syslog*, 11) *user*, 12) *uucp*, and are assigned a severity; 1) *Emergency*, 2) *Alert*, 3) *Critical*, 4) *Error*, 5) *Warning*, 6) *Notice*, 7) *Info* or 8) *Debug*, in descending order. The logs are stored; 1) locally in files under /var/log/ or 2) sent to a remote syslog daemons of a central logs repository.

2.2 Logging Frameworks

The Apache Logging Services Project [4] created their first logging framework *log4j* [5] back in 1996. The *log4j*, a popular logging framework for Java, comes to influence other logging frameworks for other programming languages; 1) *log4php* for PHP, 2) *log4net* for C# .NET, 3) log4cxx for C++, all developed and maintain under the umbrella of Apache Logging Services Project, while different independent developers had ported *log4j* to the C, Perl, Python, Ruby, and Eiffel languages. The *log4j* and its different incarnations are by far the most widely used logging frameworks as computer application events-logger today.

The popularity of *log4j* could be due to its designed decisions by separating logging into three basic components of; 1) Categories, 2) Appenders, and 3) Layouts. The three components are designed to be used together in computer programs such as to enable developers to log messages according to message type and priority (Categories), and to control at runtime how these

messages are formatted (Layouts) and where they are reported (Appenders). By choosing different mixes of these three logging components (Categories, Appenders, Layouts), developers can install flexible logging schemes for any degree of detail in any applications.

To the rest of this paper the discussion on logging will be based on the use of Python logging module which is a part of Python Standard Library since Python 2.3 version of the language. The logging module (heavily influenced by java.util.logging package and *log4j* was first developed in 2002 [6].

2.3 Python logging module

The concept of hierarchy is important in logging, it provides structure and precedence of what were logged. Thus, by their attributes of structures and precedences, logs can be successfully analyzed and effectively used by different stake-holders at the different phases of a software life-cycle.

Python logger is used in a program by importing the logging module; import logging. The classes and corresponding methods defined by the logging module, are listed below:

1. Logger objects are instantiated by getLogger(), which exposes the *logging* interface to be used by applications.
   logger = logging.getLogger()
2. Handlers send the log records to the appropriate destination.
   handler = logging.FileHandler('app.log')
3. Formatters associated with handles define the layout of log records.
   handler.setFormatter(logging.Formatter('%(name)-12s: %(levelname)-8s %(message)s'))
4. Filters provide a finer grained facility for determining which log to be recorded or not.
   logger.addFilter(filter_func())
5. Loggers are ready to be used when handles are attached to them.
   logger.addHandler(handler)

A call to getLogger() will create a logger object with a default name of *root*. Multiple calls to getLogger('zoo') return a reference to the same (in this example, *zoo*) logger object. getLogger() supports category hierarchy naming, where getLogger('zoo') is a parent to getLogger('zoo.fish') and ancestor to getLogger('zoo.fish.herring').

The instance property propagate (default value is True) controls the propagation logging messages upward (if logger.propagate == True) from child loggers to the higher level (ancestor) loggers, i.e. *zoo* will also handle a message happened in *zoo.fish.herring*. Where for instance, *zoo.fish.herring* concerns with local logging while *zoo* handles a central logging process, see Fig. 2.

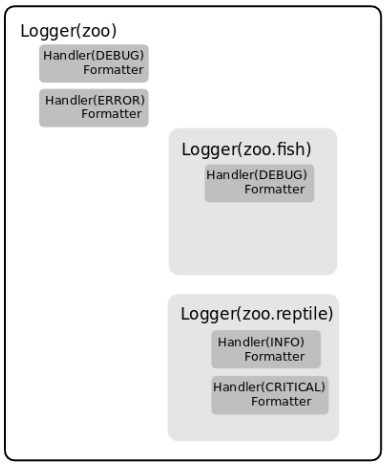

Fig. 2 Logger Hierarchal Structure

The Python logging module supports a wide variety of handler; 1) StreamHandler, 2) FileHandler, 3) WatchedFileHandle, 4) RotatingFileHandler, 5) TimedRotatingFileHandler, 6) SocketHandler, 7) DatagramHandler, 8) HTTPHandler, 9) SysLogHandler, 10) NTEventLogHandler, 11) SMTPHandler, 12) BufferingHandler, and 13) NullHandler. Normally 1) sends logging messages to console via *stderr* output stream. Logging reports are sent and stored in files by 2), 3), 4) and 5). Logging reports are sent to network servers to be appropriately handled at the servers by 6), 7) and 8). Logs are handled by standard OS (Operating System) 9) Unix/Linux or 10) Windows. Logs can also be sent via email to system administrators by 11). Logs can be temporarily buffered in core memory before being flashed to other handlers by 12). Furthermore, developers can provide costume handler whenever needed, for instance, saving logging reports to databases.

Beside %(message)s, logging.Formatter() take a variety of predefined attributes; 1) asctime, 2) created, 3) filename, 4) funcName, 5) levelname, 6) levelno, 7) lineno, 8) module, 9) msecs, 10) name, 11) pathname, 12) process, 13) processName, 14) relativeCreated, 15) thread, and 16) threadName. One singularly important attribute here is *name*, where the value is the string parameter of getLogger() instantiator, but of course to be useful a log must contain the minimum information of 1) asctime, 5) levelname, 10) name, and message.

All logging messages are associated with level of severity. The Python logging module defines these severity levels; 1) DEBUG, 2) INFO, 3) WARNING, 4) ERROR, 5) CRITICAL, ordered in increasing severity. To each logger object, a logger logging level can be set by

e.g. logger.setLevel(logging.DEBUG). The default *root* logger has logging level set at DEBUG. In addition, one can also define logging level in handler by e.g. error_hander.setLevel(logging.ERROR). Note that a logger object can have multiple handlers.

The DEBUG defined the low-water mark in severity. A logger set to DEBUG will handle all logs at all levels of severity. Only *critical* events will be logged if the logger is set to CRITICAL, all the rest of logging messages will be ignored. This mechanism gives a convenient way to developer in planning logging i relation to software life-cycle; 1) DEBUG during Development, 2) INFO or WARNING during Testing, and 3) ERROR in Deployment. This logging level can be conveniently changed in global configuration file of an application.

3 Implementation

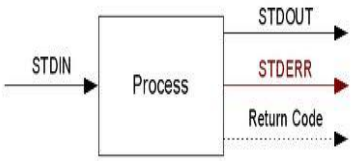

Fig. 3 Unix/Linux stdin, stdout, stderr

Traditionally in Unix/Linux OS, program interface with its operational environment through 1) *stdin* for input, 2) *stdout* for output, and 3) *stderr* for error, are defaulted to the console where the program was invoked, see Fig. 3. Developers use print to statement for debugging programs under development. While this practice is sufficient (barely), but not belonging to good practices, for very small programs, it should be avoided for large applications.

3.1 Programs, Errors, Logging

Logging has to be taken seriously. It should be an integral part of the application and not being left as an afterthought. Logging as security must be designed and implemented in parallel with and in relation to business logic of the application. Time and resources need to be provided under design and development for 1) business logic, 2) security, and 3) logging. Access to logs must be provided to all stake-holders in deployment.

An application under execution reads input X from its input interface, processes the inputs in accordance to the predetermined logic I defined within its program whereby its internal states V are changed accordingly, producing output Y to be written to its output interface, $(Y,V)_{n+1} = I(X,V)_n$. Each iteration will produce new internal state.

There are a number of sources for error; 1) input failures, 2) logic errors, 3) inconsistence internal states, 4) output failures. In order to mitigate risk of errors for higher execution stability and increase application robustness in relation to changes in its operating environment, a good mechanism for catching these exceptions [7] and logging such events must be implemented in the application. It is not uncommon that an application constitutes of many cooperating sub-programs sharing some part of applications internal states in persistence storage of multiple databases. Therefore, a *multifaceted* and *multileveled* logging strategy, Fig. 4, is necessary when deploying such a distributed application.

3.2 Challenges

Hallmarks of enterprise-wide services are centralized storage and repositories, together with a single point of access services. What this means is that enterprise-wide logging services need to consolidate logging messages from diverse sources into one central storage wich supports and facilitates easy and secure access to graded logging information. For such a system, a centralized database can be established to store all logging messages from logging clients, while a corresponding Web-based server can be deployed for anywhere and anytime access to stored logging information, Fig. 5.

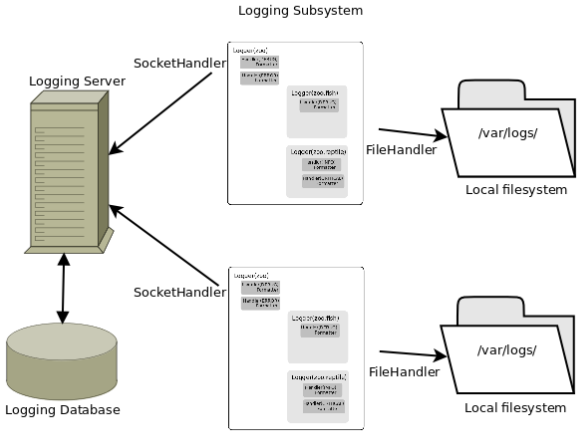

Fig. 4 Multifaceted and Multileveled Logging Scheme

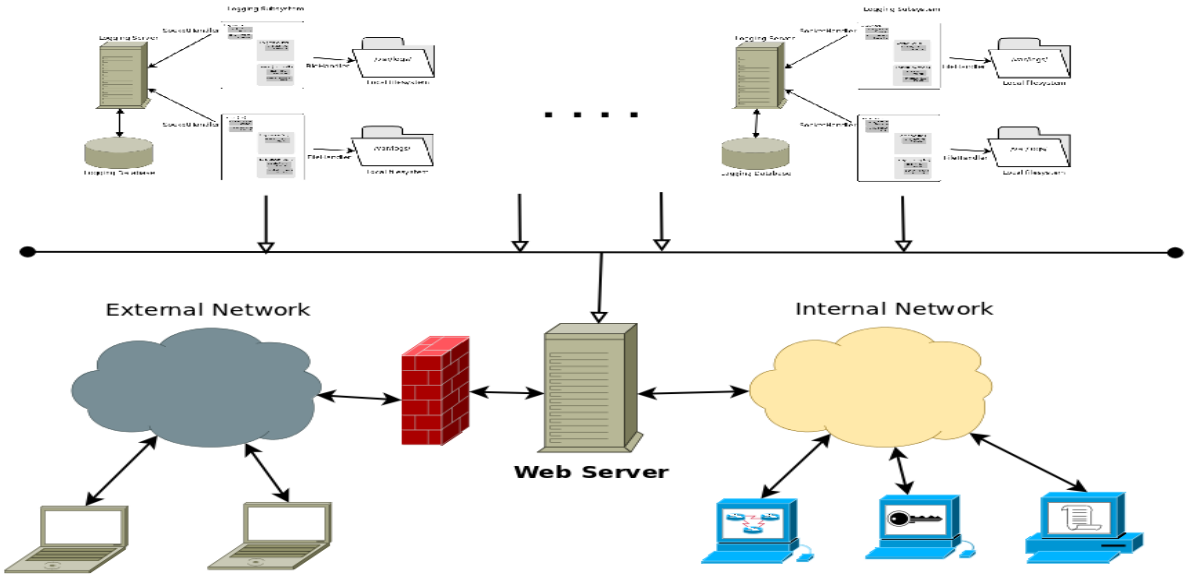

Fig. 5 Enterprise-Wide Logging System

Considering that applications logs can contain sensitive information, access control to logging reports is a matter important. Local file-based logging storages (directories and files) are easy to make secure using security provisions provided by the OS, in the form of ownership, group-ownership and others together with theirs permissions on these file resources. A centralized storage access control mechanism can be a security challenge, groups and memberships, role and permissions, have to be managed on the enterprise scale [8].

3.4 A Case Study

In this section we provide examples with simple Python codes with the supports of the appropriate Python modules (Standard libraries) on how to implement a *multifaceted* and *multileveled* logging subsystem in any application as shown schematically in Fig. 4 using 1) SocketServer and 2) sqlite3 on the server side and 3) logging and 4) logging.handlers on the client side. The diagram in Fig. 4 shows two logging subsystems implemented in two independently executing applications sending logging messages over the TCP/IP (Transmission Control Protocol/Internet Protocol) network to a single logging processing server, as well as storing logging messages into theirs local log files for local copies.

In Appendix A, we provide a simple working example of a logging processing server. The multithreading TCP/IP socket server, LogReceiver, using Python SocketServer. ThreadingTCPServer module, will listen on port 8888 for request from clients. A new thread is created when a request is detected by select.select and the request will be handled by LogHandler. The handler which is a SocketServer.StreamRequestHandler will collect all logging message block and process it into Python dictionary logging message object. Some of the selected values of this dictionary object are then save into a database, in this case a SQLite file-based database.

Since a request is handled in its own thread, each handling process needs to create an sqlite3 connection instance. The logging message data is packed using pickle by the logger on the client before it was sent over the net to the server. The pickle.loads unpacked the data block into a dictionary typed variable. The logging data provides these information; 1) msecs, 2) args, 3) name, 4) thread, 5) created, 6) process, 7) threadName, 8) module, 9) filename, 10) levelno, 11) processName, 12) pathname, 13) lineno, 14) exc_text, 15) exc_info, 16) funcName, 17) relativeCreated, 18) levelname, and 19) msg.

Developers can choose which ever of these value to be stored in to the database, e.g. the exc_text contains error messages originating form an exception, should this be considered as an important piece of information. Our example in Appendix A only created, levelname, name, filename, pathname, and msg are stored into the database.

In Appendix B, we provide simple client codes as examples. These codes are meant to be useful beyond the point of showing the usage patterns of logging client. The example codes also show hierarchal structure of Python logging module. Here, the top most logging object is created by rootLogger = logging.getLogger( ), and this logger object will have a default name *root*. We then

created two subordinate logger objects aLogger = logging.getLogger(myapp.area1) and bLogger = logging.getLogger(myapp.area2). They are subordinate objects in the sense that logging message assigned to them will also propagate upward in the hierarchy. The propagation of logging messages is controlled by the logger propagate attribute (set to True by default).

The handler of rootLogger is set to TCP/IP socket handler by rootLogger.addHandler(socketHandler) where socketHandler = logging.handlers.SocketHandler(serv, port). All logging messages to the *root* logger will be sent to our logging processing server (129.177.9.71:8888) as shown in Appendix A. A logging.FileHandler() will be used to store local copies of logging messages, where the handler will write logs to a specific log-file, in this case here, log_cln_2.log. Associated with the handler is a formatter, logging.Formatter(), in the example of Appendix B, these information will be saved; 1) created, 2) name, 3) levelname, and 4) msg + exc_text.

Since propagate attribute to aLogger and bLogger were set to True and rootLogger.setLevel(logging.DEBUG) all logging messages written to these logger will be sent to rootLogger and be saved in a log database at the central logging processing server. If we have rootLogger.setLevel(logging.ERROR) instead then logging messages at lower severity, i.e. 1) DEBUG, 2) INFO, and 3) WARNING will not be processed by the server.

Appendix C shows a listing of logging messages get logged in local log-file log_cln_2.log and at central logging database logs.db. The listing form log-file shows line format corresponding to the format declared in logging.Formatter(). Lines listed form database were raw data rows as defined. Be aware that exc_text is not recoded in the database since it is not defined in the data model.

Through a software life-cycle many stake-holders are collaborating to archive preset goals. Sharing logging reports within an organization and across organizational boundaries within a cooperative framework will give invaluable informational asset in working toward the goals. The best way to share logging rapports stored in multiple databases would be coupling them with a Web-based logging distribution services or applications as shown in Fig 5. Any numbers of databases can be exposed to the Web server. A flexible scheme of access controls could be implemented both for users within the organization and collaborators from outside of organization, where the Web server will act as a portal to all logging processing servers.

4. Conclusions

All developers had in one way or anther done logging during development process. Any piece of program is sprinkled with some print statements, as a way to diagnose the program under development, these will print diagnostic massages to console (i.e. stdout). When a program becomes large and complicated these practice will become difficult to manage. One can see diagnostic print statements for development purposes left in the release version of the program that can create intermittent problems which are difficult to trace and fix under deployment.

It is probably best with logger.setLevel (logging.DEBUG) during development and adjust the logging level to, e.g. logger.setLevel(logging.ERROR) for deployment. We can still have logger.debug(msg) or logger.info(msg) in the code, but need worrying the effect of streaming massages to stdout during deployment since with logging level set to ERROR, logger.debug(msg) or logger.info(msg) will be silently filter out by the logging module. When unknown errors occur during deployment and detail diagnostic is needed the logging level can then be set back to DEBUG with eventually some new logger.debug(msg) statements.

Software logging concerns with; 1) What (notes, records, massages), 2) Where (function, module, host, process), 3) Why (info, errors, exceptions), and 4) When (timestamps). To do useful logging one needs to ask and answer these questions; 1) What happened?, 2) When did it happen?, 3) Where did it happen?, 4) Why did it happen, and 5) How important is it? The Python logging module implements a logging framework with all those in mind, and employing logging library in one program will help to manage logging effectively. By coupling logging to databases with a Web-based distribution application, a complete, flexible, effective, and manageable enterprise-wide logging can be accomplished.

Appendix A

Python Logging Server Example

```
1import SocketServer
2
3class LogHandler(SocketServer.StreamRequestHandler):
4
5 def handle(self):
6 import struct
7 import pickle
8 import sqlite3
9 while True:
10 chunk = self.connection.recv(4)
11 if len(chunk) < 4:
12 break
```

```
13 slen = struct.unpack('>L', chunk)[0]
14 chunk = self.connection.recv(slen)
15 while len(chunk) < slen:
16 chunk = chunk + self.connection.recv(slen - len(chunk))
17 obj = pickle.loads(chunk) # unpickle
18 con = sqlite3.connect('logs.db')
19 with con:
20 cur = con.cursor()
21 cur.execute("INSERT INTO Logs\
22 (ts, level , name , filename , pathname , msg) VALUES \
23 (:ts, :level , :name , :filename , :pathname , :msg)", \
24 {"ts": obj['created'], "level":obj['levelname'], \
25 "name":obj['name'], \
26 "filename":obj['filename'], "pathname":obj['pathname'], \
27 "msg":obj['msg'] }
28 )
29 con.commit()
30 con.close()
31
32class LogReceiver(SocketServer.ThreadingTCPServer):
33
34 def __init__(self , host='0.0.0.0', port=8888,
35 handler=LogHandler):
36 SocketServer.ThreadingTCPServer.__init__(self , (host
, port), handler)
37 self.abort = 0
38 self.timeout = 1
39 self.logname = None
40
41 def processLog(self):
42 import select
43 abort = 0
44 while not abort:
45 rd, wr, ex = select.select([self.socket.fileno()],
46 [], [],
47 self.timeout)
48 if rd:
49 self.handle_request()
50 abort = self.abort
51
52def main():
53 server = LogReceiver()
54 server.processLog()
55
56if __name__ == '__main__':
57 main()
```

Appendix B

Python Logging Client Example

```
1import logging , logging.handlers
2
3rootLogger = logging.getLogger('')
4rootLogger.setLevel(logging.DEBUG)
5socketHandler =
logging.handlers.SocketHandler('129.177.9.71', 8888)
6
7# don't bother with a formatter , since a socket handler
sends the event as
8# an unformatted pickle
9rootLogger.addHandler(socketHandler)
10
11# Now, we can log to the root logger , or any other
logger. First the root...
12logging.info('Start testing client.')
13
14# Now, define a couple of other loggers which might
represent areas in your
15# application:
16
17aLogger = logging.getLogger('myapp.area1')
18bLogger = logging.getLogger('myapp.area2')
19hndl = logging.FileHandler('log_cln_2.log')
20fmtr = logging.Formatter('%(asctime)s - %(name)s -
%(levelname)s : %(message)s')
21hndl.setFormatter(fmtr)
22
23aLogger.addHandler(hndl)
24bLogger.addHandler(hndl)
25
26aLogger.propagate = True # not need all
logger.propagate have True as default
27#bLogger.propagate = True
28
29aLogger.critical('Someone is doing bad thing to the
system !!!')
30
31man = {}
32try:
33 pos = man['John']
34except:
35 bLogger.exception("Who's John?")
```

Appendix C

Logger reports

```
% cat log cln 2.log
2012-11-26 20:53:26,425 - myapp.area1 - CRITICAL :
Someone is doing bad thing to the system !!!
2012-11-26 20:53:26,425 - myapp.area2 - ERROR : Who's
John?
Traceback (most recent call last):
File "log cln 2.py", line 35, in <module>
pos = man['John']
KeyError: 'John'
% logs.db
(1353959815.184912, 'INFO', 'root', 'log cln 2.py', 'log cln
2.py', 'Start testing client.')
(1353959815.185161, 'CRITICAL', 'myapp.area1', 'log cln
2.py', 'log cln 2.py', 'Someone is
doing bad thing to the system !!!')
        (1353959815.185283, 'ERROR', 'myapp.area2',
'log cln 2.py', 'log cln 2.py', "Who_s John?")
```

References

[1] B. W. Boehm, "A Spiral Model of Software Development and Enhancement". 3rd ed. IEEE Computer, May 1988.

[2] A. A. Chuvakin, "What Every Organization Should Log and Monitor". http://www.slideshare.net/anton chuvakin/what-every-organizationshould-log-and-monitor (last accessed 2012/11/01) Wikipedia Syslog. http://en.wikipedia.org/wiki/Syslog (last accessed 2012/11/01)
[3] Apache Software Foundation, Apache Logging Services Project. http://logging.apache.org/ (last accessed 2012/11/02)
[4] Ceki Gulcu, Log4j. http://www.javaworld.com/jw-11-2000/jw-1122-log4j.html (last accessed 2012/11/02)
[5] S. Vinay (red-dove.com) and Trent Mick (activestate.com), PEP282. http://www.python.org/dev/peps/pep-0282/ (last accessed 2012/11/02)
[6] S. Tumin and S. Encheva, "Building Robust Web-based Systems by Managing Exceptions Through Logging, Reporting and Analysis". International Conference on Data Networks, Communications, Computers, pp. 73-78, 2010
[7] S. Tumin and S. Encheva, "Simplifying enterprise wide authorization management through distribution of concerns and responsibilities". WSEAS Transactions on Information Science and Applications Volum 7.(6) pp. 830-83, 2010.

Sharil Tumin is chief engineer at IT Dept., University of Bergen. His research interests are within data security, artificial intelligence, and automation.

Sylvia Encheva is professor in mathematics and informatics at Stord/Haugesund University College. Her research interests are within decision support systems, non-classical logics, and fuzzy systems.